2–3 Trees

2–3 trees are one instance of a class of data structures called balanced trees. These data structures provide an efficient worst case instantiation for the Dictionary abstract data type. Recall that a dictionary supports the operations SEARCH, INSERT, DELETE on a set of items drawn from an ordered collection U, called the *universe*. The balanced trees support each operation in worst case time $O(\log n)$, where n is the number of items currently stored in the dictionary.

A 2–3 tree is a tree in which all the leaves are at the same level and each internal node has 2 or 3 children. All the records stored in the dictionary are held in the leaves. Recall that each record includes a key, the keys being used to order the records. Traversing the leaves of the 2–3 tree in left-to-right order yields the records in sorted order. Each internal node of the 2–3 tree stores a copy of the largest key that appears in any one of the leaves below it.

A 2–3 tree which stores n nodes has a height between $\lceil \log_3 n \rceil$ and $\lfloor \log_2 n \rfloor$. So the path length from the root to any leaf is $O(\log n)$. Each operation requires the traversal of a path from the root to a leaf and back to the root and hence requires $O(\log n)$ time.

A search in a 2–3 tree proceeds in a manner very similar to that of an ordinary binary search tree.

We describe how to perform an insertion or deletion without explaining how to update the copies of keys stored at internal nodes. The details of these internal key updates is straightforward, and we leave these to the reader.

An insertion begins by performing a search to determine where the item would be located (if it were present in the tree). The item is inserted as a leaf at this location. The new leaf's parent p now may have either three or four children. If it has three children, we are done. Otherwise, we replace p by two nodes p_1, p_2, where the two leftmost children of p are placed under p_1 and the two rightmost children are placed under p_2. Of course, the left to right order of the children is maintained. This operation is called a *node partition*. This process is then repeated at each successively higher level of the tree along the path from the inserted item to the root, as required to remove nodes with four children. A special case arises if the root is replaced by two nodes; then a new root node is created; its children are the two nodes newly formed from the old root.

A deletion has a similar flavor. Again, a search is performed to find the item (stored at a leaf). This leaf is deleted. Now the parent p has either one or two children. If it has two children, we are done. Otherwise, if p has only one child, p's siblings are checked; if an adjacent sibling s has three children, s gives p one of its children; if not, s and p are merged into a single node. This process is repeated at each successively higher level of the tree along the path from the deleted item to the root, as required to remove nodes with one child. If the root ends up with one child, the root is simply removed and its sole child becomes the new root.

Remark. Instead of storing a single key at each internal node, one might store either one or two keys at that mode, as follows: in the case where the node has two children, the maximum key appearing as in a leaf of the tree rooted by the left child, and in the case where the node has three children, we additionally store the maximum key appearing as a leaf in the tree rooted by the middle child. By storing these keys at internal nodes, the search procedure may be somewhat more efficient, especially if nodes are stored as records on disk: all of the information necessary to choose which child to examine next is stored in the node itself. However, keeping this information properly maintained is just slightly more tedious.